UNIVERSITY OF NEBRASKA AT OMAHA
COURSE SYLLABUS/DESCRIPTION

Department and Course Number	CSCI 4700
Course Title	Compiler Construction
Course Coordinator	Stanley Wileman
Total Credits	3
Repeat for Credit	No
Date of Last Revision	October 23, 2008

1.0 Course Description Information

1.1 Catalog Description:
Assemblers, interpreters and compilers. Compilation of simple expressions and statements. Analysis of regular expressions. Organization of a compiler, including compile-time and run-time symbol tables, lexical scan, syntax scan, object code generation and error diagnostics.

1.2 Prerequisites of the course
1.2.1 CSCI 3320
1.2.2 CSCI 3660
1.2.3 CSCI 4220
1.2.4 CSCI 4500 (recommended)
1.2.5 CSCI 4660 (recommended)

1.3 Overview of content and purpose of the course
This course is an introduction to the purpose, design, and implementation of tools for language – in particular, programming language – translation. There are two major categories of such tools: compilers and interpreters. Compilers translate constructs in a “high level” language (e.g. C++ or Java) into a language more closely related to the architecture of a computer system (e.g. assembler language or machine language). The result of this translation – normally called “object code” – is usually manipulated with other tools (library utilities, and linkers) to produce an executable program for a particular operating system and computer architecture. Interpreters, on the other hand, convert the “high level” language construct into an internal form which is the used as the input to a program that simulates the behavior of a virtual system suitable for the execution of programs in that “high level” language. There is a continuum of translation approaches. Pure compilers translate to a specific machine language. Pure interpreters are often capable of being used on arbitrary systems. Modern Java systems typically are hybrid systems, translating the Java program into “byte code” for a virtual machine and then interpreting that byte code, and perhaps translating it to machine language in some cases for more efficient execution.

Obviously there is a great deal of content that can be addressed in this course. The major phases of a compiler (lexical analysis, syntax analysis, semantic analysis, error handling, code optimization, and code generation) will be treated, including discussion of the common data structures (in particular, symbol tables) and ancillary tools (e.g. linkers). Interpreters will be considered as "simplified" compilers.

Traditional programming languages (e.g. C, C++, and Java) are not the only languages to which the techniques discussed in this course apply. A large number of additional tools that are "language driven" (e.g. awk, sed and troff) use the same techniques. To the extent time permits, the use of the techniques covered in the course for such tools is considered.

Traditional tools used to construct lexical analyzers and parsers (UNIX lex and yacc) are commonly used in this course. However, other similar tools could be used, and indeed are likely to be discussed in the course.

1.4 Unusual circumstances of the course.
None

2.0 Course Justification Information

2.1 Anticipated audience / demand

This is an elective course primary for computer science majors and others interested in the design and implementation of compilers, interpreters, and languages for computer execution. The demand is typically relatively low, and is in the same category as other courses related to low-level system programming.

2.2 Indicate how often this course will be offered and the anticipated enrollment.

The course will be offered no more frequently than once each academic year. Demand may dictate that it not be offered that frequently.

2.3 If it is a significant change to an existing course, please explain why it is needed.

3.0 List of performance objectives stated in learning outcomes in a student's perspective.

1. Describe the organization and function of each major component of an interpreter and a compiler for a simple high-level language (e.g. C or Pascal). Specifically, include the following:
    - lexical analysis,
    - parsing (syntactic analysis),
    - semantic analysis,
    - error treatment,
    - optimization (of several types),
    - code generation, and
    - block-structured symbol tables.
2. Construct lexical analysis routines for reasonable languages, including the use of tools like LEX, and explain the interaction between the lexical analyzer and the parser. This should include the ability to handle language features like simple macros and included source files.
3. Discuss and compare various approaches to parsing, including at least recursive descent, LL, and LR, and LALR techniques. The student should be able to demonstrate these parsing techniques on simple sentences from a small grammar.

4. Discuss various techniques for associating semantics with syntactical constructs, including direct generation of object code and the use of intermediate representations (e.g. triples and quads). Be able to indicate reasons why multi-pass compilation may be desirable or required.
5. Be able to use a tool like YACC to assist in the production of a parser for a reasonable high-level language.
6. Describe the basic types of optimization that may be performed by a compiler, including local and global optimization, and optimization on intermediate code as compared with optimization on emitted machine-language code. Be able to discuss and illustrate with examples topics like unrolling loops, identifying invariant expressions, optimizing the use of registers, selecting alternate instructions, and selecting alternate addressing modes for operand access.
7. Discuss the organization of the symbol table for a compiler, including the effect of block-structured languages on the symbol table. Be able to describe the content of the symbol table entry, including things like identifier lifetime, allocation (static, dynamic, stack), data type, reference variables and pointers, procedure parameters, initialization, and so forth. The significance of the symbol table as a central data structure in the compiler should be clear.
8. Be able to demonstrate the effect of various declarative statements in typical high-level languages. In particular, make clear those declarative features that result in executable code generation.
9. Explain the typical runtime storage administration for a high-level language (static and dynamic storage allocation, stack frames, effects of block structure and recursion, etc.). Be able to describe in detail the memory resources used by a simple program in a traditional high-level language.
10. Demonstrate the compilation of expressions including references to scalars, array elements, and structure/record components, and type coercion/casts. The difference between compile-time and run-time expressions should be clear.
    Show techniques for compiling typical control structures, including nested control structures.
11. Explain techniques for dealing with various kinds of errors, primarily including syntax errors and semantic errors.
12. Demonstrate compilation techniques for handling procedure definitions, declarations and invocations. In particular, discuss the treatment of parameters in reasonable detail, including parameter checking (number and type), type coercion, parameter passing, and stack frame and activation record management. Make clear the handling of function results and the compilation of external procedures.
13. Discuss the use of libraries. In particular, describe typical uses of intrinsic functions by a compiler.
14. Demonstrate basic familiarity with various object module formats, including knowledge of things like symbol table information (for debuggers and linkers) and relocation information. Discuss the processing performed by an object module linker.
15. Presented with description of the syntax and semantics of a small high-level programming language and details of a target architecture, discuss various approaches to the design of a translator for the language. After design decisions have been made, be able to implement the design to yield a complete, working translator.

4.0 Content and Organization Information

		Contact hours
4.1	Introduction and overview of the compilation process	1.5
4.2	Lexical analysis	3.0
	4.2.1 The task of the lexical analyzer	
	4.2.2 Manually recognizing lexemes	
	4.2.3 Automating the process: LEX	
4.3	Syntax analysis (parsing)	7.0
	4.3.1 Grammars	
	4.3.2 Top-down approaches	
	4.3.3 Predictive parsers	
	4.3.4 Bottom-up approaches	
	4.3.5 Automating the task: YACC	
4.4	Semantic processing	10.0
	4.4.1 Semantic actions	
	4.4.2 Intermediate representations of the program	
	4.4.3 Bottom-up approaches	
	4.4.4 Top-down approaches	
	4.4.5 Type checking and coercion	
4.5	Symbol tables	3.0
	4.5.1 Block-structured symbol tables	
	4.5.2 Symbol table content	
	4.5.3 Handling structures and records	
4.6	Statement-specific translation techniques	10.0
	4.6.1 Declarative statements	
	4.6.1.1 Scalar variables	
	4.6.1.2 Arrays	
	4.6.1.3 Structures and records	
	4.6.1.4 Initialization issues	
	4.6.1.5 External declarations	
	4.6.2 Expressions	
	4.6.3 Control structures	
	4.6.3.1 Goto statements and statement labels	
	4.6.3.2 Conditional statements	
	4.6.3.3 Repetition statements	
	4.6.4 Procedures and functions	
	4.6.4.1 Review of parameter passing techniques	
	4.6.4.2 Handling parameters and local declarations	
	4.6.4.3 The activation record (stack frames)	
	4.6.4.4 Procedure and function invocation	
4.7	Optimization Techniques	3.5
	4.7.1 Machine-independent optimizations	
	4.7.1.1 Basic blocks	
	4.7.1.2 Local optimization	
	4.7.1.3 Global optimization	
	4.7.2 Data flow analysis	

4.7.3 Machine-dependent optimizations
4.7.3.1 Register use optimization
4.7.3.2 Instruction and addressing mode selection
4.7.3.3 Peephole optimization

4.8 Code generation 7.0
4.8.1 Generation ignoring context
4.8.2 Context sensitive generation
4.8.3 Register use
4.8.4 Intrinsic functions
4.8.5 Examples
4.8.6 Object file formats

5.0 Teaching Methodology

5.1 Methods to be used

The primary teaching methods will be lectures and demonstrations.

5.2 Student role in the course

The student will attend lectures and demonstrations, participate in discussion on assigned readings, complete assigned projects, and complete required tests and examinations.

6.0 Evaluation Information

6.1 Types of student projects

Programming assignments, short tests, and major examinations will be used to evaluate student performance. Programming assignments are ideally designed to allow students to produce a functioning compiler for a small language defined by the instructor (e.g. a subset of Pascal or C) at the end of the semester. Each assignment targets a particular compiler component and/or a tool (e.g. a lexical analyzer is constructed using LEX, and a parser is constructed using YACC). Code is generated for a popular architecture, like a Java virtual machine or an Intel x-86 processor. Tests are used to evaluate student understanding of material which time constraints prohibit students from using in their programming assignments. Examinations cover all course material.

6.2 Basis for determining the final grade

Component	Weight
Programming assignments (the compiler)	50%
Short tests	20%
Mid-term examination	15%
Final examination	15%

6.3 Grading type

Letter grades will be determined using the weighted average of the various items used to evaluate students. A typical grade mapping is illustrated below.

Points	Grade
97-100%	A+
90-96%	A

87-89%	B+
80-86%	B
77-79%	C+
70-76%	C
67-69%	D+
60-66%	D
0-59%	F

7.0 Resource Material

7.1 Textbooks and/or other required readings used in course.

7.1.1 Aho, Lam, Sethi, Ullman, *Compilers: Principles, Techniques and Tools* (2nd edition), Addison Wesley (2006)

7.1.2 Notes on the use of LEX and YACC (frequently included in a textbook)

7.2 Other suggested reading material

None

7.3 Other sources of information

None

7.4 Current bibliography of resources for student's information

7.4.1 Aho and Ullman, *Principles of Compiler Design*, Addison Wesley (1979)

7.4.2 Appel, *Modern Compiler Implementation in Java* (2nd edition), Cambridge University Press (2002)

7.4.3 Fischer and LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings (1991)

7.4.4 Fraser and Hanson, *A Retargettable C Compiler: Design and Implementation*, Addison Wesley (1995)

7.4.5 Gries, *Compiler Construction for Digital Computers*, John Wiley (1971)

7.4.6 Holub, *Compiler Design in C*, Prentice-Hall (1990)

7.4.7 Kaplan, *Constructing Language Processors for Little Languages*, John Wiley (1994)

7.4.8 Levine, *Linkers & Loaders*, Morgan Kaufmann (2000)

7.4.9 Mak, *Writing Compilers and Interpreters: An Applied Approach Using C++* (2nd edition), John Wiley (1996)

7.4.10 McKeeman, Horning and Wortman, *A Compiler Generator*, Prentice Hall (1970)

7.4.11 Metsker, *Building Parsers with Java*, Addison Wesley (2001)

7.4.12 Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann (1997)

7.4.13 Parsons, *Introduction to Compiler Construction*, Computer Science Press (1992)

7.4.14 Welsh and Hay, *A Model Implementation of Standard Pascal*, Prentice Hall (1986)

7.4.15 Wirth, *Compiler Construction*, Addison Wesley Longman (1996)

8.0 Other Information

8.1 Accommodations statement:

8.2 Other:

8.3 Author(s): Bill Mahoney

9.0 **Computer Science Accreditation Board (CSAB) Category Content (class time in hours)**

CSAB Category	*Core*	*Advanced*
Data structures		7
Computer organization and architecture		5
Algorithms and software design		23
Concepts of programming languages		10

10.0 **Oral and Written Communications**

Every student is required to submit at least __0___ written reports (not including exams, tests, quizzes, or commented programs) to typically _____ pages and to make _____ oral presentations of typically _____ minutes duration. Include only material that is graded for grammar, spelling, style, and so forth, as well as for technical content, completeness, and accuracy.

11.0 **Social and Ethical Issues**

No coverage.

12.0 **Theoretical content**

		Contact hours
12.1	Lexical analysis	2
12.2	Grammars	2
12.3	Parsing	4
12.4	Data flow analysis	1
12.5	Semantics	6

13.0 **Problem analysis**

The core problems in programming language translation and fundamental approaches to these problems are presented. Students should be able to examine the description of a language feature and propose various alternatives for their translation.

14.0 **Solution design**

Once the approach to a language translation problem has been decided, students should be able to use the tools and techniques studied in this course to produce simple translators.

CHANGE HISTORY

Date	***Change***	***By whom***	***Comments***
06/13/2003	Cleanup	Wileman	
10/23/2008	Review and update with mapping table.	Mahoney	There’s likely a newer revision of this (post-2003) but I located this one on the CS web site; I reviewed the whole thing and made changes were appropriate, so this is the most recent for the class.
6/16/2015	Revised the syllabus format; modified the language of learning objectives.	Wileman	
6/26/2015	Numbered list of performance objectives; changed coordinator	Wileman	